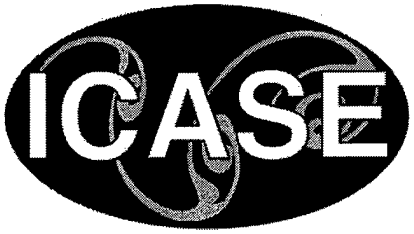
NASA/CR-2000-210088
ICASE Report No. 2000-13




# Parallel Performance Investigations of an Unstructured Mesh Navier-Stokes Solver

*Dimitri J. Mavriplis*
*ICASE, Hampton, Virginia*

*Institute for Computer Applications in Science and Engineering*
*NASA Langley Research Center*
*Hampton, VA*

*Operated by Universities Space Research Association*


National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

March 2000

# PARALLEL PERFORMANCE INVESTIGATIONS OF AN UNSTRUCTURED MESH NAVIER-STOKES SOLVER

DIMITRI J. MAVRIPLIS*

**Abstract.** A Reynolds-averaged Navier-Stokes solver based on unstructured mesh techniques for analysis of high-lift configurations is described. The method makes use of an agglomeration multigrid solver for convergence acceleration. Implicit line-smoothing is employed to relieve the stiffness associated with highly stretched meshes. A GMRES technique is also implemented to speed convergence at the expense of additional memory usage. The solver is cache efficient and fully vectorizable, and is parallelized using a two-level hybrid MPI-OpenMP implementation suitable for shared and/or distributed memory architectures, as well as clusters of shared memory machines. Convergence and scalability results are illustrated for various high-lift cases.

**Key words.** parallel, unstructured, multigrid

**Subject classification.** Applied and Numerical Mathematics

**1. Introduction.** The work described in this paper represents extensions and improvements to a previously described unstructured multigrid flow solver which has been used extensively for high-lift analysis [9, 11, 12]. Unstructured mesh approaches are well suited for high-lift applications where complicated geometries are most often encountered. However, in order to offset the additional computational overheads associated with unstructured meshes, and in the interest of enabling solutions on very high resolution grids for high accuracy, special attention must be devoted to producing a rapidly converging algorithm, as well as an extremely scalable solution procedure which can run efficiently on thousands of processors.

The basic solution algorithm consists of a non-linear multigrid solver, enhanced by a directional line-implicit preconditioning technique for overcoming the stiffness associated with highly-stretched meshes. In the current work, the existing unstructured multigrid solver has been extended to support both cache-based and vector architectures as well as multi-level parallelism. The original MPI-based parallel implementation has been extended to a two-level parallelization strategy which employs MPI to communicate between groups of partitioned subdomains, and OpenMP to communicate between various subdomains contained within each MPI process. In this manner, the code can be run in a purely MPI mode, suitable for distributed memory architectures, a purely OpenMP mode suitable for shared memory architectures, or a hybrid two level MPI-OpenMP mode suitable for clusters of shared memory processors, typical of many emerging large parallel supercomputer architectures.

A GMRES procedure is also introduced as an option to speed up convergence when additional memory is available. This approach is particularly attractive for medium size problems running on distributed memory architectures, where unused memory on the individual processors represents a resource which can be exploited at little extra cost by the GMRES algorithm.

*Institute for Computer Applications in Science and Engineering (ICASE), Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681–2199, U.S.A., dimitri@icase.edu. This research was partially supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-2199. Partial support was also provided under AFOSR grant AFOSR-PO-99-0005 and under U.S. Department of Energy subcontract B347882 from Lawrence Livermore National Laboratory.

One of the principal aims of this work is to provide an efficient production capability for mid-size problems (up to $10^7$grid points) on various cost-effective computer architectures, while at the same time demonstrating the feasibility of computing very large cases (ultimately up to $10^9$ grid points) on custom parallel supercomputers such as those currently being developed for the DOE ASCI program.

**2. Base Solver.** The Reynolds averaged Navier-Stokes equations are discretized by a finite-volume technique on meshes of mixed element types which may include tetrahedra, pyramids, prisms, and hexahedra. In general, prismatic elements are used in the boundary layer and wake regions, while tetrahedra are used in the regions of inviscid flow. All elements of the grid are handled by a single unifying edge-based data-structure in the flow solver [13].

The governing equations are discretized using a central difference finite-volume technique with added matrix-based artificial dissipation. The matrix dissipation approximates a Roe Rieman-solver based upwind scheme [19], but relies on a biharmonic operator to achieve second-order accuracy, rather than on a gradient-based extrapolation strategy [8]. The thin-layer form of the Navier-Stokes equations is employed in all cases, and the viscous terms are discretized to second-order accuracy by finite-difference approximation. For multigrid calculations, a first-order discretization is employed for the convective terms on the coarse grid levels.

The basic time-stepping scheme is a three-stage explicit multistage scheme with stage coefficients optimized for high frequency damping properties [25], and a CFL number of 1.8. Convergence is accelerated by a local block Jacobi preconditioner, which involves inverting a $5 \times 5$ matrix for each vertex at each stage [18, 14, 15, 16]. A low-Mach number preconditioner [27, 23, 26] is also implemented. This has been found to be essential for high-lift flows which may contain large regions of low Mach number flow particularly on the lower surfaces of the wing. The low-Mach number preconditioner is implemented by modifying the dissipation terms in the residual as described in [8], and then taking the corresponding linearization of these modified terms into account in the Jacobi preconditioner, a process sometimes referred to as "*preconditioning*$^2$" [8, 24].

The single equation turbulence model of Spalart and Allmaras [22] is utilized to account for turbulence effects. This equation is discretized and solved in a manner completely analogous to the flow equations, with the exception that the convective terms are only discretized to first-order accuracy.

**3. Directional-Implicit Multigrid Algorithm.** An agglomeration multigrid algorithm [7, 13, 21] is used to further enhance convergence to steady-state. In this approach, coarse levels are constructed by fusing together neighboring fine grid control volumes to form a smaller number of larger and more complex control volumes on the coarse grid. A multigrid cycle consists of performing a time-step on the fine grid of the sequence, transferring the flow solution and residuals to the coarser level, performing a time-step on the coarser level, and then interpolating the corrections back from the coarse level to update the fine grid solution. The process is applied recursively to the coarser grids of the sequence.

While agglomeration multigrid delivers very fast convergence rates for inviscid flow problems, the convergence obtained for viscous flow problems remains much slower, even when employing preconditioning techniques as described in the previous section. This slowdown is mainly due to the large degree of grid anisotropy in the viscous regions. A directional smoothing technique [8, 9] is employed to overcome this aspect-ratio induced stiffness. Directional smoothing is achieved by constructing lines in the unstructured mesh along the direction of strong coupling (i.e., normal to the boundary layer) and solving the implicit system along these lines using a tridiagonal line solver.

A weighted graph algorithm is used to construct the lines on each grid level, using edge weights based on the stencil coefficients for a scalar convection equation. This algorithm produces lines of variable length. In regions where the mesh becomes isotropic, the length of the lines reduces to zero (one vertex, zero edges), and the preconditioned explicit scheme described in the previous section is recovered. An example of the set of lines constructed from the two-dimensional unstructured grid in Figure 3.1 is depicted in Figure 3.2.

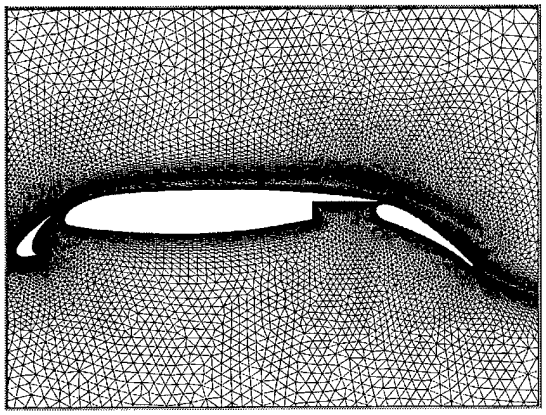


FIG. 3.1. *Unstructured Grid for three-element airfoil; Number of Points = 61,104, Wall Resolution* $= 10^{-6}$ *chords*

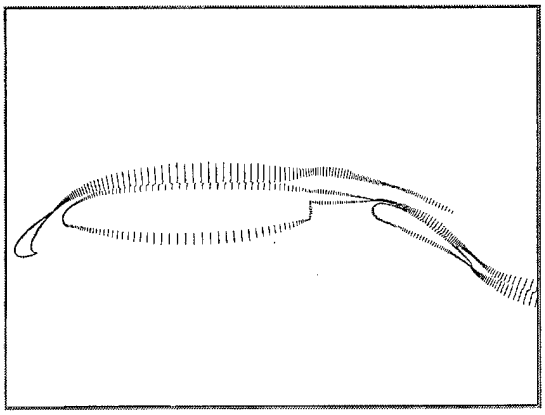


FIG. 3.2. *Directional Implicit Lines Constructed on Grid of Figure 3.1 by Weighted Graph Algorithm*

**4. Domain Decomposition.** The unstructured multigrid solver is parallelized by partitioning the domain using a standard graph partitioner [5, 6], allocating one or more partitions to each processor of a parallel computer or cluster of interconnected machines, and communicating between the various grid partitions using either the MPI message-passing library [4] or the OpenMP shared memory protocols [2].

In the multigrid algorithm, the vertices on each grid level must be partitioned across the available processors. Since the mesh levels of the agglomeration multigrid algorithm are fully nested, a partition of the fine grid could be used to infer a partition of all coarser grid levels. While this would minimize the communication in the inter-grid transfer routines, it affords little control over the quality of the coarse grid partitions. Since the amount of intra-grid computation on each level is much more important than the inter-grid computation between each level, it is essential to optimize the partitions on each grid level rather than between grid levels. Therefore, each grid level is partitioned independently. This results in unrelated coarse and fine grid partitions. In order to minimize inter-grid communication, the coarse level partitions are renumbered such that they are assigned to the same processor as the fine grid partition with which they share the most overlap. For each partitioned level, the edges of the mesh which straddle two adjacent processors are assigned to one of the processors, and a "ghost vertex" is constructed in this processor, which corresponds to the vertex originally accessed by the edge in the adjacent processor (c.f. Figure 4.1). During a residual evaluation, the fluxes are computed along edges and accumulated to the vertices. The flux contributions accumulated at the ghost vertices must then be added to the flux contributions at their corresponding physical vertex locations in order to obtain the complete residual at these points. This phase incurs interprocessor communication. In an explicit (or point implicit) scheme, the updates at all points can then be computed without any interprocessor communication once the residuals at all points have been calculated. The newly updated values are then communicated to the ghost points, and the process is repeated.

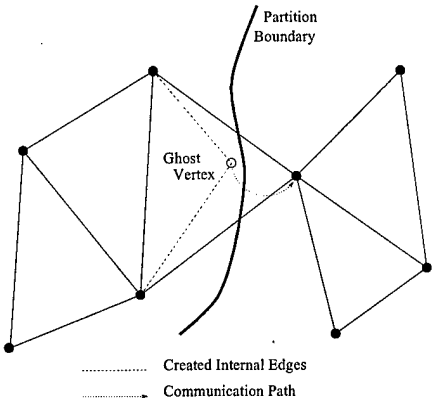



FIG. 4.1. *Illustration of Creation of Internal Edges and Ghost Points at Inter-processor Boundaries*

The use of line-solvers can lead to additional complications for distributed-memory parallel implementations. Since the classical tridiagonal line-solve is an inherently sequential operation, any line which is split between multiple processors will result in processors remaining idle while the off-processor portion of their line is computed on a neighboring processor. However, the particular topology of the line sets in the unstructured grid permit a partitioning the mesh in such a manner that lines are completely contained within an individual processor, with minimal penalty (in terms of processor imbalance or additional numbers of cut edges). This can be achieved by using a weighted-graph-based mesh partitioner such as the CHACO [5] or MeTiS [6] partitioners. Weighted graph partitioning strategies attempt to generate balanced partitions of sets of weighted vertices, and to minimize the sum of weighted edges which are intersected by the partition boundaries.

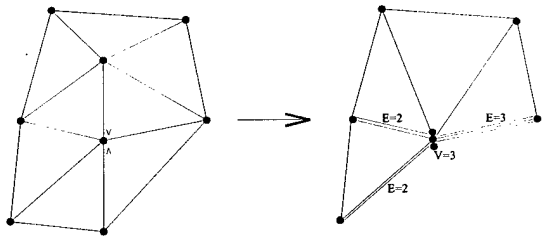



FIG. 4.2. *Illustration of Line Edge Contraction and Creation of Weighted Graph for Mesh Partitioning; V and E Values Denote Vertex and Edge Weights Respectively*

In order to avoid partitioning across implicit lines, the original unweighted graph (set of vertices and edges) which defines the unstructured mesh is contracted along the implicit lines to produce a weighted graph. Unity weights are assigned to the original graph, and any two vertices which are joined by an edge which is part of an implicit line are then merged together to form a new vertex. Merging vertices also produce merged edges as shown in Figure 4.2, and the weights associated with the merged vertices and edges are taken as the sum of the weights of the constituent vertices or edges. The contracted weighted graph is then partitioned using one of the partitioners described in references [5, 6], and the resulting partitioned graph is then de-contracted, i.e., all constituent vertices of a merged vertex are assigned the partition number of

that vertex. Since the implicit lines reduce to a single point in the contracted graph, they can never be broken by the partitioning process. The weighting assigned to the contracted graph ensures load balancing and communication optimization of the final uncontracted graph in the partitioning process.

Due to the large size of the grids considered in this work, all preprocessing operations must be performed on a large parallel supercomputer. This includes the agglomeration procedure, the partitioning of the various coarse and fine multigrid levels, and the determination of the inter-processor communication schedules. This is mostly due to the large memory requirements of these procedures, (which run between 50% and 75% of the memory requirements of the flow solver, i.e., 1 Kbyte per grid point), rather than the CPU time requirements, which are small compared to those of the flow solver. At present, these procedures are executed sequentially on a single processor of an SGI Origin 2000, but using large portions of the memory of the entire machine. For example, the various preprocessing operations for a 24.7 million point grid required between 10 to 20 Gbytes of memory and between 45 minutes to 90 minutes for each of the operations mentioned above. The sequential execution of large jobs of this nature is made possible by the shared memory architecture of the SGI ORIGIN 2000, and cannot be performed on purely distributed memory machines or on clustered machine architectures. The complete parallelization of these procedures for distributed-memory machines is currently under development.

**5. Cache-Optimization and Vectorization.** On each partitioned domain, the solver must be optimized for the processor architecture to which the domain is assigned. The two types of architectures supported are cache-based scalar microprocessors, and vector processors. For a cache-based scalar microprocessor, grid vertices and edges are reordered to increase locality and hence cache efficiency. This is done individually on each partition. The grid points are reordered using a breadth-first search technique, similar to a Cuthill-McKee [3] reordering strategy. The edges are then reordered so that all edges touching each (reordered) vertex and not previously listed are ordered sequentially,

For vector processor architectures, the grid vertices are reordered in the same manner as described above, but the edges must be sorted into groups, such that within each group no two edges access the same vertex, in order to prevent data-recurrences. Vectorization can then proceed within each group. Since many current vector architectures include a memory cache, the reordering of vertices for locality can still be beneficial. The block tridiagonal line solves are vectorized by grouping the lines into sets of 64 or 128, which are then vector-processed. For lines of unequal length, this involves padding the shorter lines with identity matrices in order to achieve a group of uniform line length.

Figure 5.1 illustrates the single processor computational rates achieved for a small problem (a grid of approximately 200,000 points) on various current processors. A computational rate of 225 Mflops is achieved on the Cray-SV1 vector processor, while a rate of 75 Mflops is achieved on an Origin 2000 processor (250 MHz R10000).

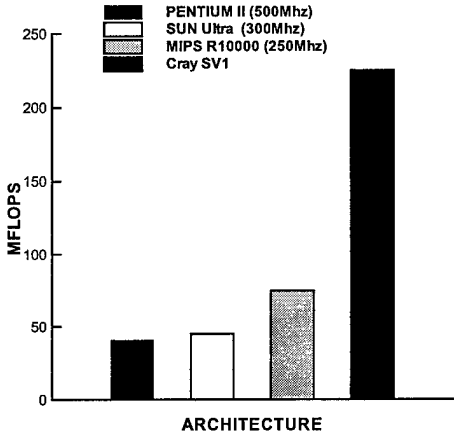



FIG. 5.1. *Computational Rates of Unstructured Multigrid Solver on Various Processors*

**6. Parallel Programming Models.** For parallel execution, the partitioned subdomains are distributed to the various processors of a parallel machine or a cluster of machines. During the parallel execution of the program, inter-processor communication between the ghost points and their real images in neighboring partitions is required (see Figure 4.1). For distributed memory architectures, this communication is implemented using the MPI message-passing library [4]. The inter-processor communication patterns are pre-determined at run-time. Communication is then executed by packing messages from all ghost points on a given processor that are to be sent to another processor into a buffer that is then sent as a single message. This standard approach to inter-processor communication has the effect of reducing latency overheads by creating fewer larger messages.

For shared memory architectures, a potentially more efficient communication strategy is to simply copy (or copy-add) the values from the individual ghost points into the locations which correspond to their real images, since the memory on different partitions is addressable from any other partition. Additionally, the OpenMP standard [2] provides a simple strategy for parallelizing shared memory programs through the use of compiler directives.

The original parallel implementation of the unstructured mesh solver was written using the MPI communication library [11]. The solver has currently been extended to include the capability for running OpenMP in the place of, or in addition to, MPI. This is achieved by first modifying the code to enable the sequential processing of multiple subdomains on each processor. This involves wrapping a loop over the number of subdomains on a processor around the original subroutines which performed the computations in the MPI program. Parallelization over the local subdomains can then be achieved simply by inserting the appropriate OpenMP compiler directive directly preceding the loop over the number of subdomains. In addition, routines which identify the memory locations of the ghost vertices and their corresponding real images in neighboring subdomains must be constructed, as well as the routines which actually copy these values to and from their corresponding locations.

For each MPI process, the individual arrays are initialized globally across all local subdomains, and pointers which identify the starting location of each subdomain are constructed. These arrays, indexed by the subdomain pointer, are then passed as arguments to the subdomain subroutines.

```
include OMP_DIRECTIVE
do : Loop over number of partitions
   call domain_local_routine(array(ptr(partition_id)))
-------------------------------------------------
   do : Loop over number of vector groups
     do : Loop over edges in a vector group
       n1 = edge_end(1,iedge)
       n2 = edge_end(2,iedge)
       flux = function of values at n1,n2
       residual(n1) = residual(n1) + flux
       residual(n2) = residual(n2) - flux
     enddo
   enddo
-------------------------------------------------
 enddo
c
include OMP_DIRECTIVE
do : Loop over number of partitions
   call OMP_communicate
enddo
c
include OMP_DIRECTIVE
do : Loop over number of partitions
   call MPI_communicate
enddo
```

FIG. 6.1. *Pseudo Code Illustration of the Code Structure for Vectorized Hybrid MPI-OpenMP Routine for Thread-to-Thread Communication Model. (Dashed lines delimit in-lined subroutine representation)*

In this manner, all array references in the subdomain routines are subdomain-local, and the existing MPI subdomain code is preserved. Figure 6.1 illustrates the code structure within an MPI process. The initial loop runs over the number of local subdomains. Since all these loops throughout the code are similar, the whole code can be parallelized under OpenMP using a handful of compiler directives. The subdomain routine is called in this primary loop. For brevity, the subdomain routine code has been inlined in the pseudo code of Figure 6.1. This includes the second loop and third loops of the figure. The second loop runs over the sorted groups of edges in order to enable vectorization within a group. For a scalar processor, the number of such groups reduces to one, which includes all edges. The third loop runs over the edges within a group. After these three nested loops are executed, the routine which performs the shared memory communication is called, followed by the routine which performs MPI communication, in the case of the thread-to-thread communication model described below.

The code structure is such that no explicit OpenMP synchronization steps (*omp barrier*) are employed. Rather, separate parallelized do loops are employed. While these loops contain implicit synchronization steps, they also enable the sequential execution of the code in the absence of any OpenMP directives. This enables the code to run with multiple partitions on individual processors.

The current implementation results in a code which can be run in a purely MPI mode, suitable for distributed memory architectures, a purely OpenMP mode, suitable for shared memory architectures, or a two-level hybrid MPI-OpenMP mode, suitable for clusters of shared-memory processors.

There are various possible strategies for implementing MPI communication in conjunction with OpenMP.

While OpenMP achieves parallelism by spawning multiple threads within a process [2], the MPI library is only defined on a process basis and in general cannot distinguish between multiple threads. However, in a thread-safe MPI implementation [1], MPI calls may be executed by the individual threads in a multi-threaded environment. A communication strategy which can be executed entirely in parallel consists of having individual threads perform MPI calls to send and receive messages to and from other threads living on other MPI processes, as illustrated in Figure 6.2. In this case, the MPI calls must specify the process identifier (*id number*) as well as the thread id to which the message is being sent (or received). While the specification of a process id is a standard procedure within an MPI call, the specification of a thread id can be implemented using the MPI send-recv tag [4]. In this approach, the size and number of messages is identical to that produced by an equivalent code running MPI alone on all subdomains.

An alternate approach, illustrated in Figure 6.3, consists of having all threads pack their messages destined for other threads of a particular remote MPI process into a single buffer, and then having the MPI process (i.e., the master thread alone) send and receive the message using MPI . The received messages can then be unpacked or scattered to the appropriate local subdomains. This packing and unpacking of messages can be done in a thread-parallel fashion. However, the MPI sends and receives are executed only by the master thread, and these operations may become sequential bottlenecks since all other threads remain idle during this phase. One way to mitigate this effect is to overlap OpenMP and MPI communication. Using non-blocking sends and receives, the master thread first issues all the MPI receive calls, followed by all the MPI send calls. After this, while the MPI messages are in transit, the OpenMP communication routines are executed by all threads, after which, the master thread waits until all MPI messages are received. Thread-parallel unpacking of the MPI messages then proceeds as usual. This approach also results in a smaller number of larger messages being issued by the MPI routines, which may be beneficial for reducing latency on the network supporting the MPI calls. On the other hand, there is always a (thread-) sequential portion of communication in this approach, which may degrade performance depending on the degree of communication overlap achieved. Note that the grouping of communication into multiple overlapping levels is not particular to the MPI-OpenMP programming model, but could be implemented on MPI-alone or OpenMP-alone models, although implementation would be considerably more complicated.

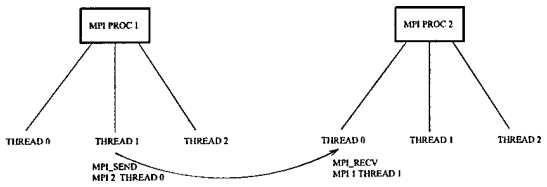



FIG. 6.2. *Illustration of Thread-to-Thread MPI Communication for a Two-level Hybrid MPI-OpenMP Implementation*

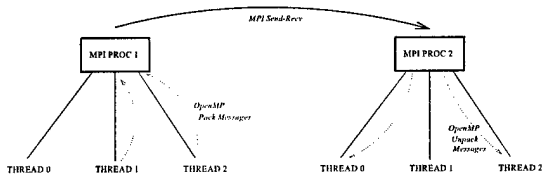



FIG. 6.3. *Illustration of Master-Thread Controlled MPI Communication for a Two-level Hybrid MPI-OpenMP Implementation*

**7. Scalability Results.** We begin with a comparison between pure MPI and pure OpenMP implementations on two shared-memory architectures, the Cray SV1, and the SGI Origin 2000, for the single grid (non-multigrid) solver. The test case consists of a 3.1 million point grid about an aircraft high-lift system with no nacelle, which has been previously described in detail [12]. For comparison purposes, the solver achieves a single processor computational rate of 75 Mflops on the Origin 2000, and 225 Mflops on the Cray SV1. Figure 7.1 depicts a comparison of the scalability achieved using OpenMP and using MPI on the Cray

SV1, while Figure 7.2 depicts the same comparison on the Origin 2000. In both cases, the two approaches yield very similar results. The Cray SV1 contains a relatively flat memory architecture and the MPI library is implemented using the shared memory protocols, so one would expect the two approaches to yield similar results. The OpenMP implementation is seen to give slightly lower scalability although the actual timings are never more than 10% apart for both methods.

The cc-NUMA memory architecture of the SGI Origin 2000 can significantly alter the performance of a shared memory implementation depending on how the requested memory is mapped to the architecture, since this memory is logically shared, but physically distributed. The current implementation makes use of the *first touch* rules, in which memory is allocated to the processors which are the first to access or touch it. Memory placement is thus achieved by executing a parallel loop in which each processor initializes all arrays on the subdomain(s) to which it has been assigned. Figure 7.2 indicates that the performance of OpenMP and MPI are very similar on the Origin 2000, right up to 128 processors. OpenMP is again slightly slower than the MPI implementation, but the timings differ by no more than 10% in all cases. The slightly slower OpenMP results may be due to the higher number of implicit synchronizations in this implementation.

Figure 7.3 illustrates the speedups achieved for a small problem (200,000 grid points) running on single and dual 400MHz Pentium II processors in a shared memory cabinet using MPI and OpenMP. In this case, the OpenMP result is slightly faster than the MPI result using the same two shared-memory processors. On the other hand, the best result is obtained using MPI on two distributed memory processors, which offers twice the effective memory bandwidth of the shared memory configuration.

Figure 7.4 depicts the relative speedups in going from 16 to 32 processors for the 3.1 million point case discussed previously on a cluster of 32 Pentium 500 MHz processors, arranged as 16 cabinets with two shared memory processors each. The baseline 16 cpu case was run using one MPI process on each processor. The 32 cpu case was run as a pure MPI code using one MPI process on each processor, and as a mixed MPI - OpenMP code, using one MPI process on each cabinet, with 2 OpenMP threads per cabinet, using both communication models described in the previous section. In this case, the MPI-alone strategy produces the best speedup, although the differences between all three methods are very small.

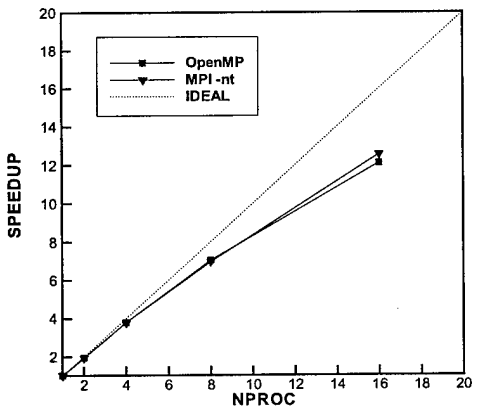



FIG. 7.1. *Observed Speedups for MPI-alone and OpenMP-alone Implementations of Single Grid Solver on Cray-SV1 Vector Machine*

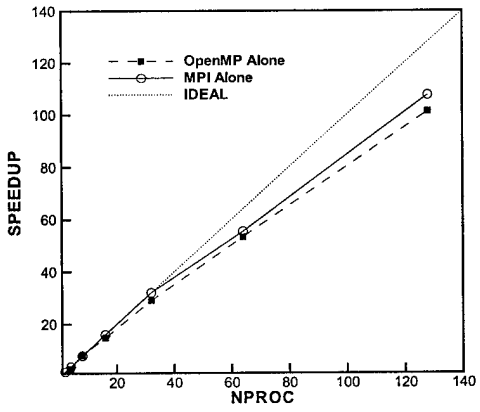



FIG. 7.2. *Observed Speedups for MPI-alone and OpenMP-alone Implementations of Single Grid Solver on SGI Origin 2000*

Figure 7.5 depicts the relative timings for the same case running in the hybrid MPI-OpenMP mode on an Origin 2000, using different combinations of MPI and OpenMP, up to 128 processors, for the thread-to-thread communication model, while Figure 7.6 depicts the results obtained using the overlapping MPI-OpenMP communication model. The data point at 64 MPI processes and 2 OpenMP threads in Figure 7.5 is not representative, since in this case the operating system would occasionally place two threads on one processor. With this exception, both figures indicate that for small OpenMP thread counts, a minor slowdown is observed for the hybrid model over the pure MPI model, with mounting efficiency losses as the number of threads is increased, although these are substantially smaller for the overlapping MPI-OpenMP communication model.

In the case of thread-to-thread communication, much of the slowdown has been traced to the MPI calls locking and thus executing sequentially at the thread level. Apparently, the definition of "thread-safe" MPI at present simply refers to the possibility of executing MPI calls from a multi-threaded environment, and does not cover the thread-parallel execution of such MPI procedures. Because there are more messages to be sent and no overlap in this case, poorer performance than in the alternate approach is observed. The performance of MPI under OpenMP can be expected to be dependent on vendor implementation, and it is still not clear whether fully thread-level parallel MPI communication will be implemented by vendors in future releases.

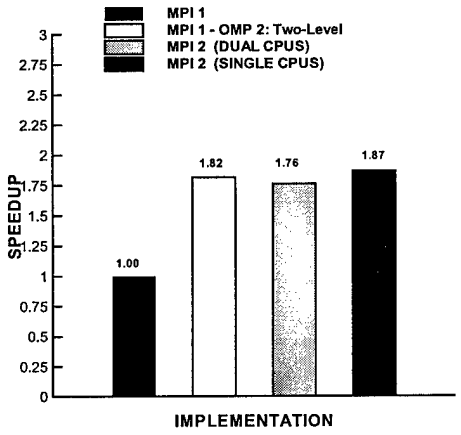



FIG. 7.3. *Relative Speedup from 1 to 2 cpus for Pure MPI and hybrid MPI-OpenMP Implementations of Single Grid Solver on a Dual Pentium II 400 MHz PC*

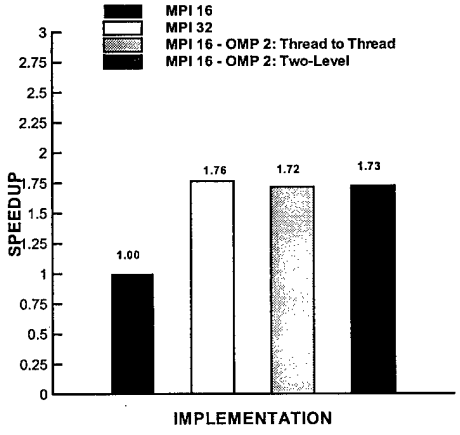



FIG. 7.4. *Relative Speedup from 16 to 32 cpus for Pure MPI and hybrid MPI-OpenMP Implementations of Single Grid Solver on Cluster of Dual Pentium III 500 MHz PCs*

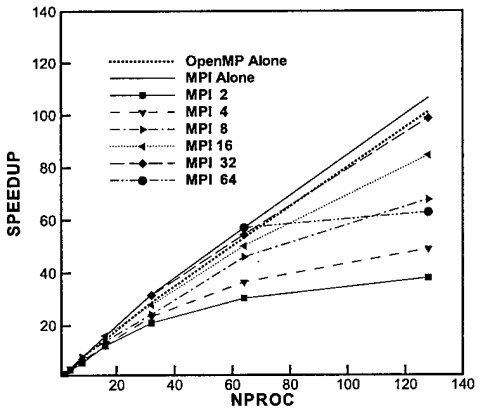



FIG. 7.5. *Observed Speedups for two-level hybrid MPI-OpenMP Implementations of Single Grid Solver on SGI Origin 2000 using Thread-to-Thread Communication Strategy*

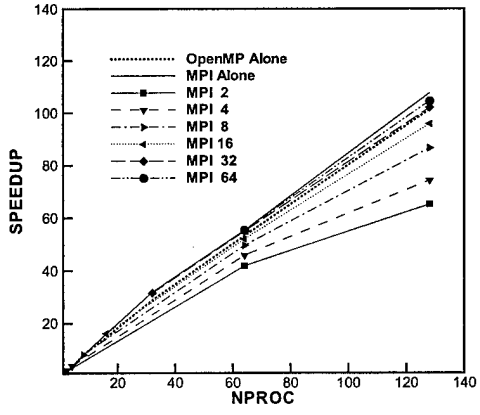



FIG. 7.6. *Observed Speedups for two-level hybrid MPI-OpenMP Implementations of Single Grid Solver on SGI Origin 2000 using Overlapping Two-Level Communication Strategy*

Figures 7.7 through 7.9 depict the scalability for the same case using MPI exclusively on three large parallel machines: the ASCI Red machine, an Intel based machine at Sandia National Laboratory, the ASCI Blue Pacific Machine, an IBM based machine at Lawrence Livermore National Laboratory, and the ASCI Blue Mountain Machine, a collection of 16 × 128 cpu Origin 2000 Machines at Los Alamos National Laboratory. In the first two cases, the scalability of the single grid solver is also compared with that of the multigrid solver using 5 grid levels. As expected, the multigrid solver delivers somewhat lower scalability than the single grid solver due to the larger amount of communication generated on the coarser grids, although both algorithms follow the same asymptotic trends. In practice, the multigrid solver always delivers much faster convergence and must be used for converging real problems. However, the single grid scalability results can be interpreted as an upper limit on the scalability achievable by the multigrid algorithms.

From these figures, the best scalability is observed on the ASCI Red machine with good speedups observed right up to 2048 processors, while scalability on the two other machines begins to drop off around 256 to 512 processors. Better scalability is observed for larger problem sizes, as shown in Figure 7.9 and Figure 7.10, where a 24.7 million point grid (exact subdivision by 8 of the previous grid) is seen to scale reasonably well up to 1024 Origin 2000 processors, and up to 1450 processors on the T3E-1200E.

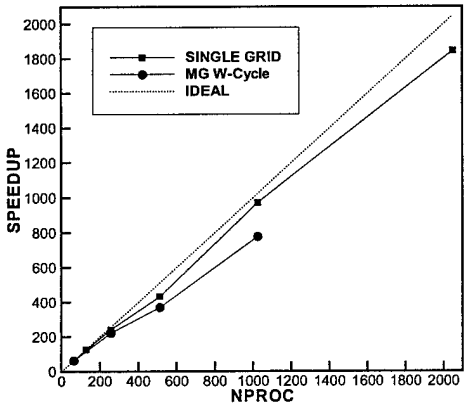



FIG. 7.7. *Observed Speedups for Single Grid and Multigrid Solver on Intel-Based Machine at Sandia National Laboratory*

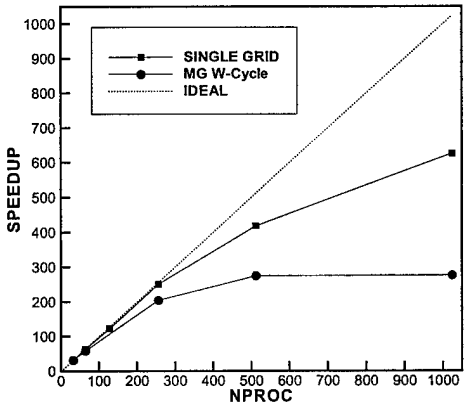



FIG. 7.8. *Observed Speedups for Single Grid and Multigrid Solver on IBM-Based Machine at Lawrence Livermore National Laboratory*

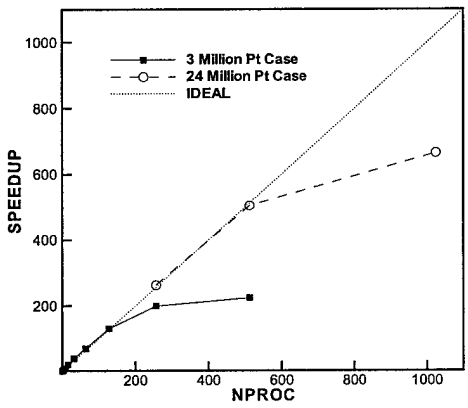



FIG. 7.9. *Observed Speedups for 3.1 million point and 24.7 million point single grid problem on ASCI Blue Mountain SGI-Origin-Cluster Machine at Los Alamos National Laboratory*

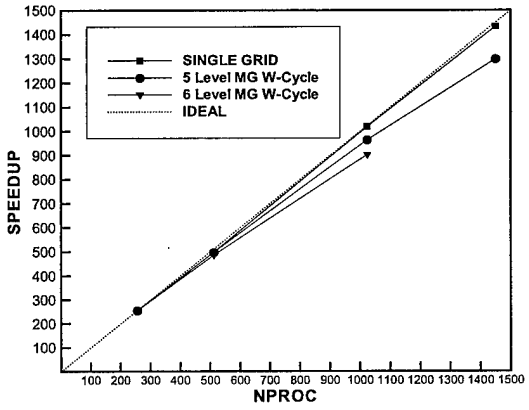



FIG. 7.10. *Observed Speedups for 24.7 million point single grid problem on Cray T3E-1200E*

**8. GMRES Acceleration.** While good scalability on large numbers of processors is important for reducing turnaround time, accelerating the numerical convergence to steady-state is equally important in achieving this goal. Although the preconditioned unstructured multigrid algorithm described previously [10] provides relatively fast convergence for many cases, further increases in convergence efficiency can be achieved by incorporating a Krylov acceleration technique such as the General Minimum Residual (GMRES) method [20]. The existing preconditioned directional implicit agglomeration multigrid algorithm can be employed as

a preconditioner itself to GMRES [8, 15]. The current implementation uses a nonlinear GMRES solver [28] which computes Jacobian-vector products by finite differencing the residual.

Parallelization of the GMRES algorithm is almost trivial, since the bulk of the work is confined to the existing parallel multigrid solver. The principal additional steps involve the computation of global norms for each search direction (implemented as parallel reduction operations), and the solution of a least-squares problem of the order of the number of search directions, which is performed redundantly on each processor.

The addition of GMRES incurs little extra cpu time, measured on a multigrid cycle basis, but requires considerable additional storage, since a solution vector must be stored for each of the Krylov search directions. In the current implementation, 20 search directions are employed, resulting in a memory increase of 100 words per vertex (about 50% increase).

One of the attractive features of this implementation is that the number of search directions can be specified at run time. Since many parallel computers are run in the space-sharing (as opposed to time-sharing) mode, each cpu most often hosts a single process. In situations where this process does not require the entire amount of memory local to that processor, additional GMRES search directions can be used to make use of this "free" memory, thus accelerating convergence and reducing overall cpu time.

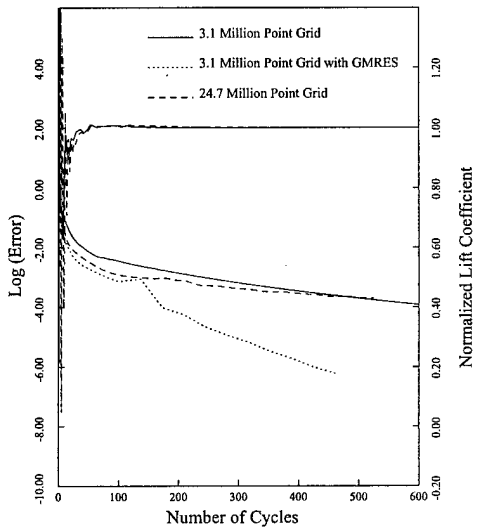



FIG. 8.1. *Multigrid Convergence Rates for Coarse (3.1 million pt) Grid with and without GMRES acceleration and Fine (24.7 million pt) Grid without GMRES acceleration at 0.2 Mach Number and 10 degrees Incidence*

The convergence history for the previously discussed aircraft high-lift case is shown in Figure 8.1. The freestream Mach number is 0.2, the incidence is 10 degrees, and the Reynolds number is 1.6 million. Convergence is shown for the coarse 3.1 million point grid with and without GMRES acceleration option, as well as for the 24.7 million point grid without GMRES. The convergence histories of the fine and coarse grids without GMRES are very similar, indicating that the multigrid algorithm is successful in providing grid independent convergence rates. The addition of GMRES in the coarse grid case (initiated after 100 multigrid cycles) is seen to accelerate substantially the asymptotic convergence. As can be surmised from this example, the addition of GMRES is most beneficial when convergence to very low tolerance levels is desirable. Note that for the fine grid, GMRES could not be applied due to the lack of available memory.

**9. Additional High-Lift Cases.** In order to demonstrate the capability of the current methodology in handling realistic complex geometries, the flow over a complete high-lift transport configuration has been computed. The baseline geometry is similar to the one discussed in the previous section, and described in more detail in previous work [12]. However, the pylon and nacelle have been added to this geometry to create a realistic full configuration high-lift case. The grid generated for this case is depicted in Figure 9.1. This grid was generated using the VGRID program [17] and contains 2.9 million vertices and 16.9 million cells, with a spacing at the aircraft surface skin of $1.35 \times 10^{-6}$ root chord lengths. A qualitative depiction of the computed solution on this grid is given in Figure 9.2 for a freestream Mach number of 0.2, an incidence of 10 degrees, and a Reynolds number of 1.6 million. The convergence rate for this case is very similar to that displayed in Figure 8.1, for the 3.1 million point grid without GMRES, and is therefore not reproduced here. The residuals were decreased by four orders of magnitude over 500 multigrid cycles, using a five level multigrid sequence with no GMRES acceleration. This case was run on a cluster of 32 Pentium II 400 MHz cpus, and required 5 Gbytes of memory and 5.5 hours to obtain the final solution. A complete comparison of these computed results with experimental wind-tunnel results is planned for the near future.

The next test case involves an experimental high-lift geometry known as the *Trapezoidal Wing* configuration. This geometry is currently the subject of an extensive experimental investigation aimed at providing a complete set of surface and off-body flow data to enable comparison and validation of CFD codes for high-lift flows. The configuration consists of a half-span low-aspect ratio swept wing, with a full span slat and full span flap. The freestream Mach number is 0.2, and the Reynolds number is 19 million based on the reference chord. A grid of 2.4 million points has been generated about this configuration, with a normal wall spacing of $2 \times 10^{-6}$ chords, and is illustrated in Figure 9.3.

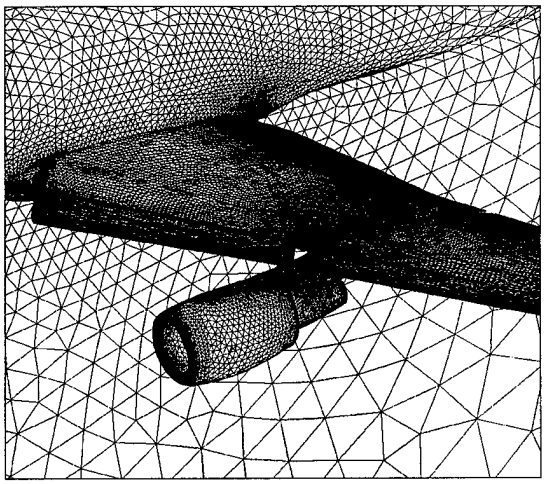


FIG. 9.1. *Unstructured Grid for Complete Wing-Body-Nacelle-Pylon Geometry; Number of Grid Points = 2.9 million*

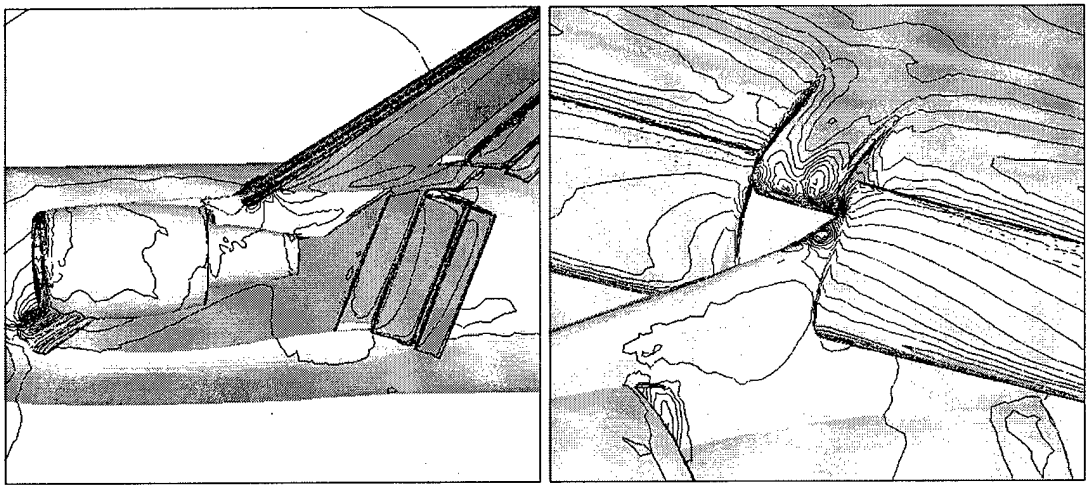


FIG. 9.2. *Computed Pressure Contours for Flow Over Complete Wing-Body-Nacelle-Pylon High-Lift Configuration*

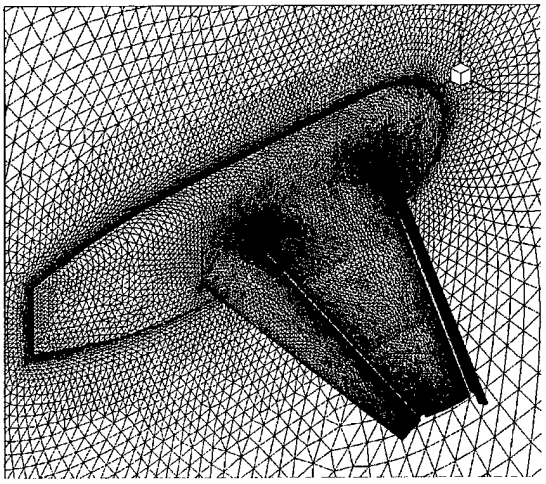


FIG. 9.3. *Unstructured Grid for Trapezoidal Wing High-Lift Geometry; Number of Grid Points: 2.8 million*

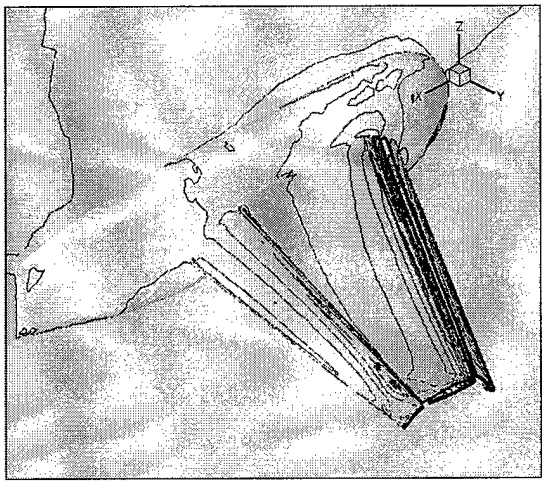



FIG. 9.4. *Computed Surface Density Contours for Flow over Trapezoidal Wing Geometry at 28 degres Incidence. Mach = 0.2, Reynolds = 19 million*

A sample solution is depicted in Figure 9.4, as computed surface density contours on the wing. The convergence history for the computed flow at 28 degrees incidence is given in Figure 9.5, using the multigrid algorithm with five grid levels. A total of 4.8 Gbytes of memory and approximately 35 minutes of wall clock time were required on a 128 cpu Origin 2000 to obtain this level of convergence.The computed lift curve is compared with experimental data in Figure 9.6. Maximum lift occurs at approximately 34 degrees, at a Cl value of about 2.8. Although the location of the $Cl_{max}$ point is relatively well predicted, the level is somewhat lower than the experimental values. This is most likely the result of insufficient grid resolution. A full grid refinement study along with a more detailed comparison of computed and experimental values for this case involving surface pressures and off-body flow profiles is planned for the near future.

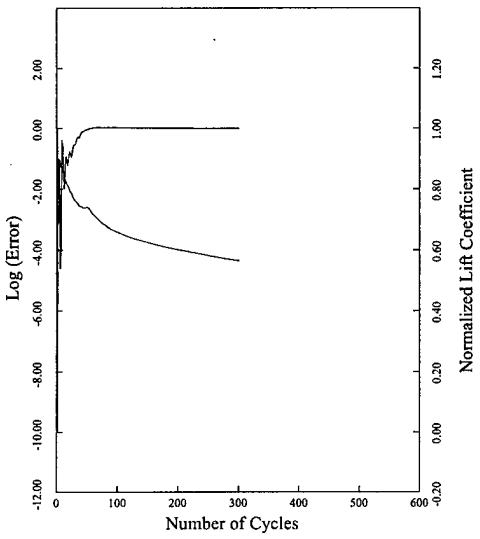



FIG. 9.5. *Convergence Rate for Trapezoidal Wing Configuration at 28 degrees Incidence using Multigrid Algorithm*

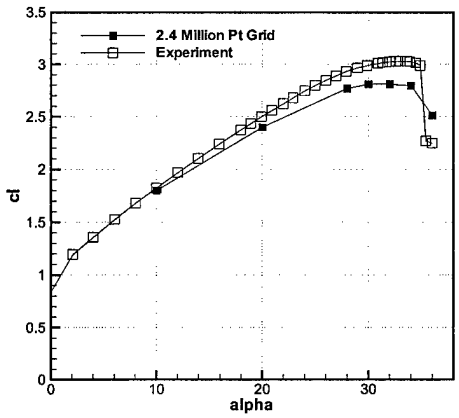



FIG. 9.6. *Computed Values of Lift Coefficient versus Incidence for Trapezoidal Wing Geometry. Mach = 0.2, Reynolds = 19 million*

**10. Conclusions.** Table 1 illustrates the computational resources required for sample high-lift analysis cases of medium and high resolution. These results indicate that unstructured mesh analyses involving several million grid points are currently efficient enough to be productionalized on cost-effective mid-size parallel computer architectures, and that very large high-resolution cases can be carried out on capable current-day supercomputers. The current parallel implementation supports MPI, OpenMP and a two-level hybrid MPI-OpenMP strategy for clusters of shared memory processors. On shared memory machines, the performance of MPI alone and OpenMP alone appear to be equivalent. Otherwise, a pure MPI-based strategy has been found to deliver better performance than hybrid combinations of MPI and OpenMP. However, these results are necessarily dependent on hardware and vendor implementation of the parallel libraries, and evaluation will continue as new hardware becomes available. Strategies which make use of unused system resources, such as a run-time specified GMRES option have also been shown to increase overall efficiency. Future work will concentrate on parallelizing the preprocessing operations such as grid partitioning and coarse level multigrid agglomeration, in order to enable the demonstration of much larger cases on available supercomputers.

TABLE 10.1

*Sample Timings for Medium and Large Problems on Various Computer Architectures. Memory Quoted in Gbytes, Solution Time Quoted in minutes for 500 multigrid cycles*

| Platform | Procs | Memory | Time |
|---|---|---|---|
| **3.1 M Points, 18 M Cells** | | | |
| Pentium (400MHz) | 32 | 5.5 | 345 |
| Origin 2000 (250MHz) | 128 | 5.5 | 75 |
| Cray SV1 | 16 | 5.5 | 205 |
| **24.7 M Points, 144 M Cells** | | | |
| T3E-600 | 512 | 52 | 235 |
| T3E-1200e | 1450 | 52 | 62 |

**11. Acknowledgements.** Special thanks are due to David Whitaker and Cray Research for dedicated computer time, and to S. Pirzadeh for generating all of the grids used in this work with the VGRID program. This work was partly performed under the auspices of the U.S. Department of Energy under subcontract B347882 from Lawrence Livermore National Laboratory and was also partially supported by the Air Force Office of Scientific Research (AFOSR) under grant AFOSR-PO-99-0005 monitored by L. Sakell.

## REFERENCES

[1] *MPI-2: Extensions to the message-passing interface.* http://www-unix.mcs.anl.gov/mpi, 1999.

[2] *OpenMP: Simple, portable, scalable SMP programming.* http://www.openmp.org, 1999.

[3] E. CUTHILL AND J. MCKEE, *Reducing the band width of sparse symmetric matrices*, in Proc. ACM Nat. Conference, 1969, pp. 157–172.

[4] W. GROPP, E. LUSK, AND A. SKJELLUM, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, MA, 1994.

[5] B. HENDRICKSON AND R. LELAND, *The Chaco user's guide: Version 2.0.* Tech. Rep. SAND94-2692, Sandia National Laboratories, Albuquerque, NM, July 1995.

[6] G. KARYPIS AND V. KUMAR, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Tech. Report Technical Report 95-035, University of Minnesota, 1995. A short version appears in Intl. Conf. on Parallel Processing, 1995.

[7] M. LALLEMAND, H. STEVE, AND A. DERVIEUX, *Unstructured multigridding by volume agglomeration: Current status*, Computers and Fluids, 21 (1992), pp. 397–433.

[8] D. J. MAVRIPLIS, *Multigrid strategies for viscous flow solvers on anisotrpic unstructured meshes*, in Proceedings of the 13th AIAA CFD Conference, Snowmass, CO, June 1997, pp. 659–675. AIAA Paper 97-1952-CP.

[9] ——, *Directional agglomeration multigrid techniques for high-Reynolds number viscous flows.* AIAA Paper 98-0612, Jan. 1998.

[10] ——, *On convergence acceleration techniques for unstructured meshes.* AIAA Paper 98-2966, presented at the 29th AIAA Fluid Dynamics Conference, Albuquerque, NM, June 1998.

[11] ——, *Threee dimensional high-lift analysis using a parallel unstructured multigrid solver*, in Proceedings of the 16th AIAA Applied Aerodynamics Conference, Albuquerque, NM, June 1998, pp. 376–390. AIAA Paper 98-2619-CP.

[12] D. J. MAVRIPLIS AND S. PIRZADEH, *Large-scale parallel unstructured mesh computations for 3D high-lift analysis.* AIAA Paper 99-0537, presented at the 37th AIAA Aerospace Sciences Meeting, Reno, NV, Jan. 1999.

[13] D. J. MAVRIPLIS AND V. VENKATAKRISHNAN, *A unified multigrid solver for the Navier-Stokes equations on mixed element meshes*, International Journal for Computational Fluid Dynamics, 8 (1997), pp. 247–263.

[14] E. MORANO AND A. DERVIEUX, *Looking for O(N) Navier-Stokes solutions on non-structured meshes*, in 6th Copper Mountain Conf. on Multigrid Methods, 1993, pp. 449–464. NASA Conference Publication 3224.

[15] C. OLLIVIER-GOOCH, *Towards problem-independent multigrid convergence rates for unstructured mesh methods i: Inviscid and laminar flows*, in Proceedings of the 6th International Symposium on CFD, Lake Tahoe, NV, Sept. 1995.

[16] N. PIERCE AND M. GILES, *Preconditioning on stretched meshes.* AIAA Paper 96-0889, Jan. 1996.

[17] S. PIRZADEH, *Unstructured grid generation for complex 3D high-lift configurations.* Paper 1999-01-5557, presented at the AIAA-SAE World Aviation Congress, San Francisco, CA, Oct. 1999.

[18] K. RIEMSLAGH AND E. DICK, *A multigrid method for steady Euler equations on unstructured adaptive grids*, in 6th Copper Mountain Conf. on Multigrid Methods, NASA conference publication 3224, 1993, pp. 527–542.

[19] P. L. ROE, *Approximate Riemann solvers, parameter vectors and difference schemes*, J. Comp. Phys., 43 (1981), pp. 357–372.

[20] Y. SAAD AND M. H. SCHULTZ, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856–869.

[21] W. A. SMITH, *Multigrid solution of transonic flow on unstructured grids*, in Recent Advances and Applications in Computational Fluid Dynamics, Nov. 1990. Proceedings of the ASME Winter Annual Meeting, Ed. O. Baysal.

[22] P. R. SPALART AND S. R. ALLMARAS, *A one-equation turbulence model for aerodynamic flows*, La Recherche Aérospatiale, 1 (1994), pp. 5–21.

[23] E. TURKEL, *Preconditioning methods for solving the incompressible and low speed compressible equations*, J. Comp. Phys., 72 (1987), pp. 277–298.

[24] ——, *Preconditioning-squared methods for multidimensional aerodynamics*, in Proceedings of the 13th AIAA CFD Conference, Snowmass, CO, June 1997, pp. 856–866. AIAA Paper 97-2025-CP.

[25] B. VAN LEER, C. H. TAI, AND K. G. POWELL, *Design of optimally-smoothing multi-stage schemes for the Euler equations.* AIAA Paper 89-1933, June 1989.

[26] B. VAN LEER, E. TURKEL, C. H. TAI, AND L. MESAROS, *Local preconditioning in a stagnation point*, in Proceedings of the 12th AIAA CFD Conference, San Diego, CA, June 1995, pp. 88–101. AIAA Paper 95-1654-CP.

[27] J. M. WEISS AND W. A. SMITH, *Preconditioning applied to variable and constant density time-accurate flows on unstructured meshes.* AIAA Paper 94-2209, June 1994.

[28] L. B. W. N. J. YU AND D. P. YOUNG, *GMRES acceleration of computational fluid dynamic codes*, in Proceedings of the 7th AIAA CFD Conference, July 1985, pp. 67–74. AIAA Paper 85-1494-CP.